Fast Phonetic Similarity Search over Large Repositories

Hegler Tissot, Gabriel Peschl, and Marcos Didonet Del Fabro

Federal University of Parana, C3SL Labs, Curitiba, Brazil
{hctissot,gpeschl,marcos.ddf}@inf.ufpr.br

Abstract. Analysis of unstructured data may be inefficient in the presence of spelling errors. Existing approaches use string similarity methods to search for valid words within a text, with a supporting dictionary. However, they are not rich enough to encode phonetic information to assist the search. In this paper, we present a novel approach for efficiently perform phonetic similarity search over large data sources, that uses a data structure called *PhoneticMap* to encode language-specific phonetic information. We validate our approach through an experiment over a data set using a Portuguese variant of a well-known repository, to automatically correct words with spelling errors.

Keywords: Phonetic Similarity, String Similarity, Fast Search

1 Introduction

A large amount of unstructured data is being produced by different kinds of information systems [8], as free text from the medical records. String similarity algorithms are used to identify concepts when text is loaded with misspellings [4]. Edit Distance (ED) [9] and Jaro-Winkler (JW) distance [14] are two well known functions that can be used to compare the elements of some input data source with an existing dictionary. Princeton WordNet (PWN) is a lexical database that provides an intuitive combination of dictionary and thesaurus to support text analysis [10]. However, PWN should be modified in order to support similarity search.

The existing string similarity algorithms coupled with a supporting dictionary may be very inefficient, in particular when the analyzed text has spelling errors [11], because they do not necessarily handle application aspects related to spelling errors. In these cases, it is necessary to use phonetic similarity metrics. Phonetics are language-dependent [12] and solutions for this sort of problems must be designed for each specific language. In addition, similarity algorithms are often slow when executed over large databases, though fast search methods have been implemented.

In this paper, we present an approach of fast phonetic similarity search (FPSS) over large repositories. First, we define a novel string similarity metric. Second, we present an indexed data structure called *PhoneticMap*, which

is used by our novel fast similarity search algorithm. Finally, we integrate the previous contributions with PWN to implement the fast phonetic search. We validate our approach through an experiment in which we try to promote an automatic correction of spelling errors using the Portuguese language.

This article[1] is organized as follows: Section 2 proposes a method to search for phonetic similarities; Section 3 describes the experiments; Section 4 refers to the related work and Section 5 concludes with final remarks and future work.

2 Fast Phonetic Similarity Search

In this section we describe our approach to perform fast phonetic similarity search. We present novel string and phonetic similarity functions that uses $PhoneticMaps$ to support finding similar words in the PWN repository.

2.1 String Similarity

We present a novel algorithm to calculate string similarity. The $String_{sim}$ function illustrated in Figure 1 measures the similarity between two input strings w_1 and w_2, resulting a similarity value between 0 (completely different) and 1 (exactly equal).

in:	String w_1, String w_2
out:	Number $similarity$
1:	$g_1 \leftarrow CharsFound(w_1, w_2)$;
2:	$g_2 \leftarrow CharsFound(w_2, w_1)$;
3:	$\Omega \leftarrow 0.975$;
4:	$p_1 \leftarrow \Omega^{PositionPenalty(w_1,w_2)}$;
5:	$p_2 \leftarrow \Omega^{PositionPenalty(w_2,w_1)}$;
6:	$similarity \leftarrow avg(g_1 \times p_1, g_2 \times p_2)$;
7:	$\Upsilon \leftarrow 0.005$;
8:	$S_{MAX} \leftarrow MAX(length(w_1), length(w_2))$;
9:	$s_{min} \leftarrow min(length(w_1), length(w_2))$;
10:	if $(S_{MAX} > s_{min})$ then
11:	$b \leftarrow 1 + (S_{MAX} - s_{min}) \times \Upsilon$;
12:	$f \leftarrow ln(S_{MAX} - s_{min} + 1)$;
13:	$c \leftarrow \frac{S_{MAX} - s_{min}}{2}$;
14:	$similarity \leftarrow similarity \times (\frac{1}{(b^f)^c})$;
15:	end if;
16:	return $similarity$;

Fig. 1. $String_{sim}$: A proposed string similarity function pseudocode

[1] Extended version at `http://www.inf.ufpr.br/didonet/articles/2014_FPSS.pdf`

$String_{sim}$ function calculates the average percentage between w_1 characters found in w_2 and w_2 characters found in w_1 (lines 1–6). $CharsFound(a, b)$ return the number of characters of a found in b, not taking into account the characters' position. For each character found in a different string position, a reduction penalty is calculated based on the constant Ω (lines 3–5). $PositionPenalty(p, q)$ returns the number of characters of p found in q but not in the same string position. Penalty calculated based on Ω guarantees, for example, that strings "ba" and "baba" will NOT result a $similarity = 1$. When the lengths of both strings (S_{MAX} and s_{min}) are different, there is a result adjustment in order to provide another penalty in the similarity level, based on the difference on the length of words and the factor Υ (lines 7–15). Ω (=0.975) and Υ (=0.005) were manually adjusted after testing the proposed function in an application that searches for similar names of people and companies.

2.2 Phonetic Similarity

When considering phonemes, a straightforward string comparison of characters may not be enough. In order to support indexing phonemes for a fast search, we present a structure called *PhoneticMap* and we define the *PhoneticMap Similarity*. Given a word w, the generic function $PhoneticMap(w)$ results a PhoneticMap tuple $M = (w, P, D)$, where: w is the word itself, $P = \{p_1, p_2, \ldots, p_n\}$ is a set of n phonetic variations of word w, and $D = \{d_1, d_2, \ldots, d_n\}$ is a set of n definitions, where d_i is the definition of variation p_i. Given two *PhoneticMap*s M_1 and M_2, $PhoneticMapSim(M_1, M_2)$ is a generic function that results a similarity value (ranging from 0=different to 1=equal) between M_1 and M_2.

$PhoneticMap(w)$ and $PhoneticMapSim(M_1, M_2)$ are language-dependent. We develop two variations to support the Portuguese language. The function $PhoneticMap_{PT}(w)$ returns a map of 11 entries that encodes phonetic information. Table 1 describes a $PhoneticMap$ generated for a Portuguese word. The function $PhoneticMapSim_{PT}(M_1, M_2)$ calculates the phonetic similarity between PhoneticMaps M_1 and M_2 as the string similarity weighted average between some phonetic variations of M_1 and M_2 (Formula 1), where: a) $S_w = String_{sim}((M_1.w, M_2.w)$, and b) $S_{(i)} = String_{sim}((M_1.p_i, M_2.p_i)$. We manually adjusted weights used in $PhoneticMapSim_{PT}$, in order to give more importance to similarities of consonant phonemes.

$$\frac{1 \times S_w + 2 \times S_{(1)} + 5 \times S_{(2)} + 1 \times S_{(3)} + 3 \times S_{(5)} + 2 \times S_{(7)} + 2 \times S_{(9)}}{1 + 2 + 5 + 1 + 3 + 2 + 2} \quad (1)$$

2.3 Phonetic Search

To perform a fast phonetic similarity search (FPSS), we propose a method for indexing PhoneticMaps (using single column indexes in a relational database) and phonetically searching the words. FPSS must locate phonetically similar words in the repositories based on the indexed phoneme variations, returning not

Table 1. $PhoneticMap_{PT}$("arrematação")

Entry i	Definition d_i	Phonetic variation p_i
w	Word	*arrematação*
1	Word with no accents	*arrematacao*
2	Word phonemes	*aRematasao*
3	Vowel phonemes only	*aeaaao*
4	Vowel phonemes (reverse)	*oaaaea*
5	Consonant phonemes	*Rmts*
6	Consonant phonemes (reverse)	*stmR*
7	Articulation manner	*EABC*
8	Articulation manner (reverse)	*CBAE*
9	Articulation point	*FACD*
10	Articulation point (reverse)	*DCAF*

only similar words but also the similarity level of each one. Given a word w and a minimum desirable similarity level l, $PhoneticSearch(w, l)$ is a generic function that results a set of tuples (r, s), where r is a phonetically similar word, and s is the similarity level resulted between $PhoneticMap(w)$ and $PhoneticMap(r)$, where $s \geq l$. Similarity level ranges from 0 to 1. We develop $PhoneticSearch_{PT}$ function, an extended version of the $PhoneticSearch$ function. Figure 2 shows $PhoneticSearch_{PT}$ pseudocode[2].

$PhoneticSearch_{PT}$ returns a set of similar words in Portuguese for a given input word w, considering the minimum desirable similarity level l. Additional parameters p and s set the number of extended consonant phonemes that can be considered as prefix and suffix when searching for similar words. p and s have default values 0 (zero). When $p > 0$, then $PhoneticSearch_{PT}$ uses the reverse indexed PhoneticMaps entries to locate similar words (entries 4, 6, 8 and 10 described in Table 1). Function $DBPhoneticMapSearch(i, v, e)$ finds records in the $PhoneticMap$ table, searching for $PhoneticMap$ entry i equals to value v (exact match), or entry i like value v with up to e characters added ("like" match), when $e > 0$. $PhoneticSearch_{PT}$ results a exact match when $l = 1$ (lines 2–3). Otherwise, it creates a dataset combining results of different $DBPhoneticMapSearch$ executions (line 5). In lines 6–8, phonetic variations 4, 6, 8, and 10 are used whether it is necessary to perform search over the reverse PhoneticMap entries ($p > 0$). After creating a result set of candidate words, the phonetic similarity between each found word and the search word is calculated (line 10). Words that does not satisfy the minumum similarity level l are removed from the result set (lines 10-11).

[2] The approach is presented as an instance of Portuguese language. However, it is tailored to be adapted for different languages, as English and Spanish.

```
   in: String w, Number l, Integer p, Integer s
  out: Dataset result
 1 : pm ← PhoneticMap_PT(w);
 2 : if l = 1 then
 3 :    result ← DBPhoneticMapSearch(0, pm.w);
 4 : else;
 5 :    result ←
           DBPhoneticMapSearch(1, pm.p1) ∪
           DBPhoneticMapSearch(2, pm.p2) ∪
           DBPhoneticMapSearch(3, pm.p3, s) ∪
           DBPhoneticMapSearch(5, pm.p5, s) ∪
           DBPhoneticMapSearch(7, pm.p7, s) ∪
           DBPhoneticMapSearch(9, pm.p9, s);
 6 :    if p > 0 then
 7 :       result ← result ∪
              DBPhoneticMapSearch(4, pm.p4) ∪
              DBPhoneticMapSearch(6, pm.p6, p) ∪
              DBPhoneticMapSearch(8, pm.p8, p) ∪
              DBPhoneticMapSearch(10, pm.p10, p);
 8 :    end if;
 9 :    foreach (fWord in result)
10 :       if PhoneticMapSim_PT(pm, PhoneticMap_PT(fWord))
            < l then
11 :          result.remove(fWord);
12 :       end if;
13 : end if;
14 : return result;
```

Fig. 2. $PhoneticSearch_{PT}$ pseudocode

3 Experiments

In this section we describe the experiments conducted to validate our approach. First, we compare our string similarity algorithms with two well-known ones. Second, we compare the performance of our full search method with a search using the indexed *PhoneticMap*s.

3.1 String Similarity

We performed an experiment to verify the efficiency of $String_{sim}$ in automatic error correction compared with other functions. We extracted a set of 3,933 words containing spelling errors from a sample of medical record texts in Portuguese. Each word was manually annotated with the correct spelling form (*reference* words). We used the $String_{sim}$ function to search for the 10 most similar words for each incorrect word, based on the returned similarity values. We used a Portuguese version of PWN dictionary containing 798,750 distinct words. The resultsets for each word were ranked from 1 (most similar) to 10 (less similar). We

store the rank in which each *reference* word is found in each resultset. The two previous steps were repeated using Edit Distance (ED) and Jaro-Winkler (JW) functions. Lastly, we compared the results of $String_{sim}$ against ED and JW, as shown in Table 2. $String_{sim}$ had more reference words with top-1 ranking, which is the objective of the approach. In 75.5% of cases (2,970 words), both functions find the reference word in the dictionary as a top-1 ranking (the most similar). For the remaining cases, $String_{sim}$ performs better (finds the reference word in a better rank) than ED in 16.9% of cases (666 searches with better ranking) while ED is better than $String_{sim}$ in only 5.8% (230 searches). These results are similar to those found when comparing $String_{sim}$ against Jaro-Winkler function.

Table 2. $String_{sim}$ (SS) x Edit Distance (ED)

SS Rank	ED Rank 1	2	3	4-5	6-10	*Not Found*
1	2970	420	51	30	25	26
2	127	51	37	15	18	13
3	32	12	8	8	3	7
4-5	17	8	7	4	6	3
6-10	14	1	2	4	4	0
Not Found	2	0	0	1	1	6

3.2 Full and Fast Similarity Search

We compared the performance of full and fast similarity search methods. One PhoneticMap for each PWN entry (798,750 words) and 11 single-column indexes were created – one for the $Word$ entry and one for each of the 10 phonetic variations in the $PhoneticMap$. The same set of 3,933 were used. A *Full Search* was executed – each input word was compared with each dictionary entry using the $String_{sim}$ (Figure 1), searching for words with a similarity level ≥ 0.8; the spent search time and the number of found words were computed in the result – $PhoneticMapSim_{PT}$ function was not used in the *Full Search* due to its high processing time (60 seconds in average). A *Fast Search* was executed – each input word was submitted twice to $PhoneticSearch_{PT}$, with two different set of parameters: a) similarity level ≥ 0.9, and parameters p and s both equal to 0 (similar words might have the same number of consonant phonemes); and b) similarity level ≥ 0.8, and parameters p and s both equal to 1 (similar words could have one additional consonant phonemes as prefix or suffix); *Full Search* and *Fast Search* results were compared based on the total amount of spent time to execute each search, and the number of words obtained in the result.

3.3 Comparing Results

We observed that *Fast Search* can be 10-30 times faster than *Full Search*. Although a *Full Search* is complete in terms of the resulting words, both search methods did not use the same similarity function, and they do not return the same number of similar words. Even with a different result in the fast method, $PhoneticSearch_{PT}$ is able to find the reference word for each spelling error. Table 3 compares accuracy between $String_{sim}$ (SS) and $PhoneticSearch_{PT}$ (PS). In 80.6% of cases (3.170 words), both functions find the reference word as a top-1 ranking. $String_{sim}$ performs better in 10.2% of cases (402 searches) while PS is better in 8.1% (317 searches).

Table 3. $PhoneticSearch_{PT}$ x $String_{sim}$

PS Rank	SS Rank 1	2	3	4-5	6-10	*Not Found*
1	3170	189	50	31	14	5
2	143	37	10	7	1	0
3	57	13	1	2	0	1
4-5	46	9	6	4	5	0
6-10	47	6	0	0	2	1
Not Found	59	7	3	1	3	3

4 Related Work

Edit Distance (ED) (or Levenshtein Distance) [9] calculates the minimum number of operations (single-character edits) required to transform string w_1 into w_2. ED can be also normalized to calculate a percentage similarity instead of the number of operations needed to transform one string to another. Jaro-Winkler [3] is another example of string distance function. [5] presents a survey with the existing works on text similarity. [3] compares different string distance metrics for name-matching tasks, including edit-distance like functions, token-based distance functions and hybrid methods. In addition, other examples of string similarity functions can be found in the literature, as in [7, 13, 1]. Soundex is a phonetic matching scheme initially designed for English that uses codes based on the sound of each letter to translate a string into a canonical form of at most four characters, preserving the first letter [15]. As the result, phonetically similar entries will have the same keys and they can be indexed for efficient search using some hashing method. However, Soundex fails to consider only the initial portion of a string to generate the phonetic representation, which impairs the phonetic comparison when words have more than 4-5 consonants [6]. Fast Similarity Search [2] is an ED-based algorithm designed to find strings similarities in a large database.

5 Conclusions and Future Work

We presented an approach of fast phonetic similarity search coupled with an extended version of the WordNet dictionary. Our main contribution is the definition of an indexed structure, called *PhoneticMap* that stores phonetic information to be used by a novel string similarity search algorithm. The experiments showed that the algorithm has good precision results and that it executes faster than one version not using th *PhoneticMap*. We also presented a string similarity algorithm based on the notion of penalty. We plan to use our solution to address the problem of dealing with spelling errors in an information extraction system. We also plan to explore methods to optimally tune the parameters involved in the proposed hybrid similarity metrics, and adapt it to other languages, as English and Spanish.

Acknowledgments: This work is partially financed by CAPES.

References

1. Allison, L., Dix, T.I.: A Bit-String Longest-Common-Subsequence Algorithm. In: IPL, vol. 26, pp. 305–310 (1986)
2. Bocek, T., Hunt, E., Stiller, B., Hecht, F.: Fast similarity search in large dictionaries. Department of Informatics, University of Zurich (2007)
3. Cohen, W.W., Ravikumar, P., Fienberg, S.E.: A comparison of string distance metrics for name-matching tasks. In: IIWeb, pp. 73–78 (2003)
4. Godbole, S., Bhattacharya, I., Gupta, A., Verma, A.: Building re-usable dictionary repositories for real-world text mining. In: CIKM, pp. 1189-1198. ACM (2010)
5. Gomaa, W.H., Fahmy, A.A.: A Survey of Text Similarity Approaches. In: IJCA, vol. 68, pp. 13–18. Foundation of Computer Science, New York (2013)
6. Hall, P.A.V., Dowling, G.R.: Approximate String Matching. In: ACM Comput. Surv., vol. 12, pp. 381–402. New York (1980)
7. Hamming, R.: Error Detecting and Error Correcting Codes. In: Bell System Technical Journal BSTJ, vol. 26, pp. 147–160 (1950)
8. Jellouli, I., Mohajir, M.E.: An ontology-based approach for web information extraction. In: CIST, pp. 5 (2011)
9. Levenshtein, V.I.: Binary codes capable of correcting insertions and reversals. Soviet Physics Doklady, vol. 10, pp. 707–710 (1966)
10. Miller, G.A.: WordNet: a lexical database for English. Commun. ACM, vol. 38, pp. 39–41. New York (1995)
11. Stvilia, B.: A model for ontology quality evaluation. First Monday, vol. 12 (2007)
12. Mann, V.A.: Distinguishing universal and language-dependent levels of speech perception: Evidence from Japanese listeners' perception of English. Cognition, vol. 24, pp. 169 - 196 (1986)
13. Paterson, M., Dancik, V.: Longest Common Subsequences. In: 19th MFCS, pp. 127–142. Springer (1994)
14. Winkler, William E.: String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage. In: Proceedings of the Section on Survey Research, 1990, S. 354–359
15. Zobel, J., Dart, P.W.: Phonetic String Matching: Lessons from Information Retrieval. In: SIGIR, pp. 166-172. ACM (1996)